Université Mohammed V-Agdal
Faculté des sciences de Rabat
Département de Mathématique et Informatique

Langage C

Préparé et présenté par
M. Benchrifa

Année Universitaire 2006/2007

Table de matières

- ***CHAPITRE 1*** **: Introduction**
  - Historique du langage C
  - Caractéristiques du langage C
  - Différentes phases de la programmation en C

- ***CHAPITRE 2*** **: Notions de base**
  - Premier programme en langage C
  - Composantes d’un programme en C
  - Discussion du programme premier_prog

- ***CHAPITRE 3*** **: Types de base, Opérateurs et expression**
  - Les types simples
  - Déclaration des variables simples
  - Les opérateurs standards
  - Les expressions et les instructions
  - Priorité et associativité des opérateurs
  - Les conversions de type

- ***CHAPITRE 4*** **: Lire & Ecrire des données**
  - Ecriture formatée de données : printf()
  - Lecture formatée de données : scanf()
  - Ecriture d'un caractère : putchar()
  - Lecture d'un caractère : getchar()

- ***CHAPITRE 5*** **: Structures de contrôle**
  - Structure de choix

    L'instruction if ; L'instruction d'aiguillage switch
  - Structures répétitives

    L'instruction d'itération while ; L'instruction d'itération do…while ; L'instruction d'itération for
  - Les instruction break et continue

M. Benchrifa : cours du langage C :
Filière SMI : Semestre 3 : 2006/2007

- ***CHAPITRE 6*** **: Tableaux**
  - Définition
  - Tableaux à une dimension (Vecteurs)
    - Déclaration ; Mémorisation ; …
  - Tableaux à plusieurs dimensions
    - Déclaration :
    - Tableaux à deux dimensions (matrices) :
      - *Déclaration ; Mémorisation ; ...*

- ***CHAPITRE 7*** **: Pointeurs**
  - Définition
  - Déclaration d'un pointeur
  - Opérations élémentaires sur les pointeurs
  - Pointeurs et tableaux
  - Pointeurs et tableaux à deux dimensions
  - Tableaux de pointeurs
  - Allocation dynamique de la mémoire

- *CHAPITRE 8* : **Fonctions**
  - La programmation modulaire
  -
  - Classes d'allocation
  - Définition, déclaration d'une fonction
  - Différentes sortes de variables, leur portée et leur classe d'allocation
  - Passage des paramètres d'une fonctions
  - Fonctions récursives
  - Pointeurs sur des fonctions

- *CHAPITRE 9* : **Chaînes de caractères**
  - Définition
  - Déclaration et mémorisation
  - Chaînes de caractères constantes
  - Initialisation d'une chaîne à la définition
  - Ordre alphabétique et lexicographique
  - Manipulation des chaînes de caractères
  - Tableaux de chaînes de caractères

- *CHAPITRE* 10 : **Types structures, unions, énumérés et synonymes**
  - Types structures : struct
  - Types unions : union
  - Types énumérés : enum
  - Types synonymes : typdef

- ***CHAPITRE 11*** **: Fichiers**
  - Définitions et propriétés
  - La mémoire tampon
  - Fichiers de texte et fichiers binaires
  - Fichiers standards
  - Déclaration d'un fichier
  - Ouverture et fermeture d'un fichier
  - Traitement du contenu d'un fichier
  - Détection de la fin de fichier
  - Déplacement dans le fichier
  - Gestion des erreurs
  - Quelques compléments

Chapitre 1 : Introduction

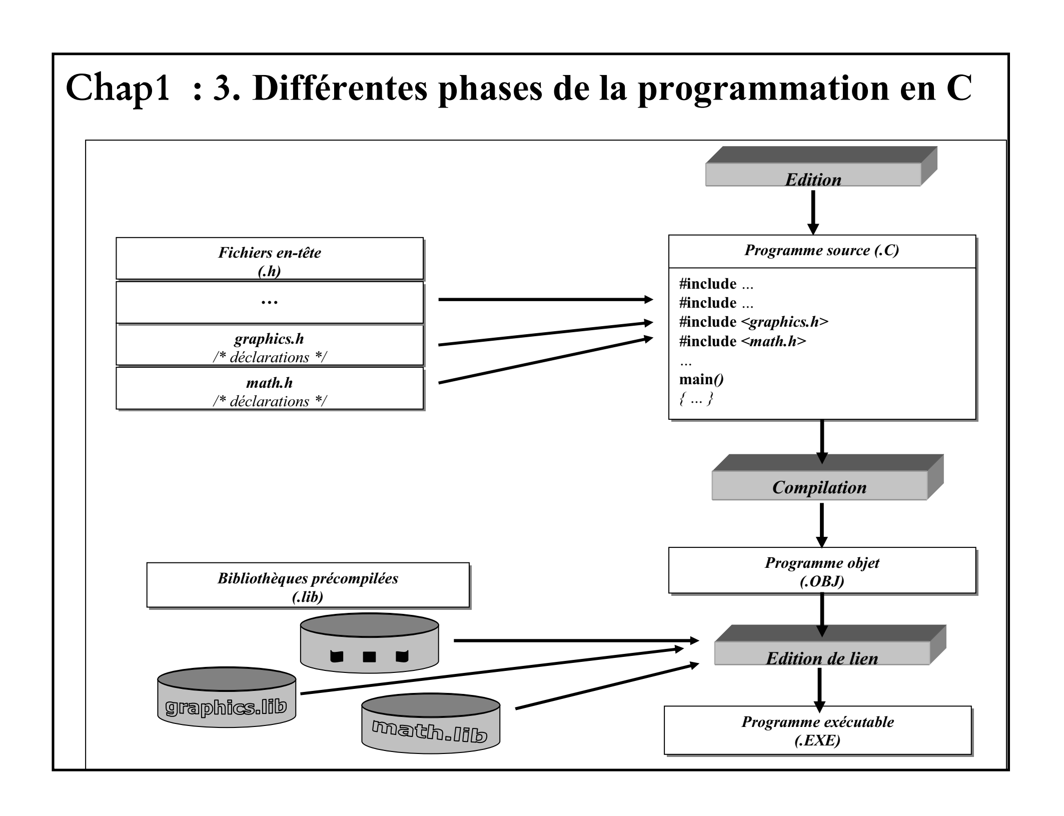
- **1. Historique du langage C**
  - **En 1972**, dans les 'Bell Laboratories', Ritchie a conçu le langage C pour développer une version portable du système d'exploitation UNIX.
  - **En 1978**, le duo Kernighan/ Ritchie a publié la définition classique du langage C,
  - **En 1983**, le 'American National Standards Institute' (ANSI) chargeait une commission de mettre au point 'une définition explicite et indépendante de la machine pour le langage C'. Le résultat était le standard Ansi-C.

Chap1 : 2. Caractéristiques du langage C

C est un langage :

- universel : permet aussi bien la programmation système que la programmation de divers applications (scientifique, de gestion, …)
- de haut niveau : C est un langage structuré (offre plusieurs structures de contrôle) et typé (déclarations obligatoires)
- prés de la machine : offre des opérateurs qui sont très proches de ceux du langage machine…
- Portable : en respectant le standard ANSI-C, il est possible d'utiliser le même programme sur d'autres compilateurs.
- …

Chap1 : 3. Différentes phases de la programmation en C



■ *CHAPITRE 2* : Notions de base

■ 1. Premier programme en langage C

```
#include <stdio.h>
int main( )
{
 printf("Bonjour tout le monde\n");
 return 0 ;
}
```

- Ce programme affiche le message :"Bonjour tout le monde".

Chap 2 : 2. Composantes d'un programme en C

En C, les programmes sont composés essentiellement de fonctions et de variables.

- **2.1 Les fonctions :**
  - Définition d'une fonction en C :

    <TypeRésultat> <Nomfonction> (<TypePar1>, <TypePar2>, …)
    {
    <déclarations locales> ;
    <instructions> ;
    }
  - En C, une fonction est définie par :
    - une ligne déclarative qui contient :
      - <TypeRésultat> : type de résultat de la fonction
      - <Nomfonction> : nom de la fonction
      - <TypePar1> <NomPar1>, … : types et noms des paramètres de la fonction
    - un bloc d'instructions délimité par les accolades {}, contenant :
      - <déclarations locales> : déclarations des données locales (c.-à-d. des données uniquement connues à l'intérieur de la fonction).
      - <instructions> : liste des instructions qui définit l'action qui doit être exécutée.
  - Remarque :
    - En C, toute instruction simple est terminée par un point virgule (;).
  - Exemple :
    - printf("Bonjour tout le monde") ;Composantes d'un programme en C

Chap 2 : 2. Composantes d'un programme en C

- **2.2 La fonction main :**
  - Une fonction et une seule s'appelle main.
  - C'est la fonction principale des programmes en C ; elle se trouve obligatoirement dans tous les programmes.
  - L'exécution d'un programme entraîne automatiquement l'appel de la fonction main.
  - Le type du résultat de main est toujours int (entier). Il n'est pas déclaré explicitement.
  - L'instruction return 0 ; indique à l'environnement que le programme s'est terminé avec succès.

Chap 2 : 2. Composantes d'un programme en C

- **2.3 Les variables** *:*
  - Contiennent les *valeurs* utilisées pendant l'exécution du programme.
  - Les noms des variables sont des **identificateurs** quelconques.
  - Toute variable doit être **déclarée** avant les instructions et son ***type*** spécifié dès la déclaration.
  - Les différents types de variables simples seront discutés dans le chapitre suivant.

Chap 2 : 2. Composantes d'un programme en C

- **2.4 Les identificateurs** :
  - Les noms des fonctions et des variables en C sont composés d'une suite de lettres et de chiffres, plus le caractère souligné (_).
  - Le 1er caractère doit être une lettre.
  - Exemples :

Identificateurs corrects :	Identificateurs incorrects :
PGCD	1PGCD
Mon_prog	Mon-prog
P1	Ptr?

  - Remarques :
    - Le caractère souligné est aussi considéré comme une lettre.
    - Ces identificateurs sont réservés :

      auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while.
    - C distingue les minuscules et les majuscules. PGCD et Pgcd sont deux identificateurs différents.

Chap 2 : 2. Composantes d’un programme en C

- **<u>2.5 Les commentaires</u> :**
  - Sont utilisés pour rendre un programme plus compréhensible.
  - Sont ignorés par le compilateur
  - un commentaire sur une ligne commence par les caractères //.
  - un commentaire multilignes commence par les caractères /* et se termine par */. A l’intérieur de ces délimiteurs, vous avez droit à toute suite de caractères (sauf évidemment */).
  - attention : on ne peut donc pas imbriquer des commentaires.

```
/************************************/
/* ce programme vous dit bonjour */
/************************************/
/*
     fichier d’entete pour utiliser la fonction printf
*/
#include <stdio.h>
/* la fonction principale main */
int main()
{
 printf(‘‘Bonjour’’); // afficher Bonjour
 return 0;
}
```


Chap 2 : 2. Composantes d’un programme en C

<u>2.6 Discussion du programme *Premier_prog*</u>

- **Le programme ne contient pas de variables, donc le bloc de déclarations est vide.**
- **la fonction main contient deux instructions :**
  - ***l'appel* de la fonction printf avec l'argument "Bonjour tout le monde\n" qui affiche le message *Bonjour tout le monde*.**
  - **L'instruction return 0 ; qui retourne la valeur 0 comme *code d'erreur* à l'environnement.**
- **La séquence de symboles '\n' ordonne un passage à la ligne suivante.**
- **La fonction printf fait partie de la bibliothèque de *fonctions standard <stdio>*, qui gère les entrées et les sorties de données.**
- **La 1ère ligne du programme #include <stdio.h> : informe le compilateur d'inclure le fichier "stdio.h" dans le texte du programme. Ce fichier contient les informations nécessaires pour utiliser les fonctions de *la bibliothèque standard <stdio>*.**

■ ***CHAPITRE 3*** **: Types de base, Opérateurs et Expressions**

■ **1. Types simples**

■ *Un type* définit l'ensemble des valeurs que peut prendre une variable, le nombre d'octets à réserver en mémoire et les opérateurs que l'on peut appliquer dessus.

■ En *C*, il n'y a que deux types de base :

- ***les entiers*** **et**
- ***les réels*.**

Chap. 3 : 1. Types simples



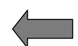
■ 1.1 Types entiers :

- 4 variantes d'entiers : *caractères* (char), *entiers courts* (short int), *entiers longs* (long int) et *entiers standards* (int).
- Caractéristiques *:*

Définition	Valeur minimale	Valeur maximale	Nombre d'octets
Char	-128	127	1
short int	-32768	32767	2
Int	-32768	32767	2
long int	-2147483648	2147483647	4

- Remarques :
  - Un caractère est un nombre entier (il s'identifie à son code *ASCII*). Par conséquent, une variable de type char peut contenir une valeur entre -128 et 127 et elle peut subir les mêmes opérations que les variables du type short, int ou long.
  - Un nombre entier de type int est souvent représenté sur *1 mot machine* (16 bits ou 32 bits). Dans notre cas, 1 mot machine = 16 bits.
  - Si l'on ajoute le préfixe unsigned à l'une de ces variantes, alors on manipule des entiers non signés :
    - **unsigned char** : indique des valeurs entières entre 0 et 255.
    - **unsigned int** (resp. short) : entre 0 et 65535.
    - **unsigned long** : entre 0 et 4294967295.

Chap. 3 : 1. Types simples



- **1.2 Types réels :**
  - 3 types de réels :
    - réels simple précision (**float**),
    - réels double précision (**double**) et
    - réels très grande précision (**long double**).
  - Caractéristiques :

Définition	*Précision*	**Répartition des bits S , M , E**	*Domaine (approximatif)*	*Nombre d'octets*
float	Simple	**1 , 23 , 8**	$\pm 1{,}1\ 10^{-38}$ à $\pm 3{,}4\ 10^{38}$	4
double	Double	**1 , 52 , 11**	$\pm 2{,}2\ 10^{-308}$ à $\pm 1{,}7\ 10^{308}$	8
long double	Très grande	**1 , 64 , 15**	$\pm 3{,}4\ 10^{-4932}$ à $\pm 1{,}1\ 10^{4932}$	10

Chap. 3 : 2. Déclaration des variables simples

Les *variables* et les *constantes* sont les données principales manipulées par un programme.

- Les variables :
  - Les *déclarations* introduisent les variables, fixent leur type et parfois aussi leur valeur de départ (initialisation).



  - Syntaxe de déclaration *:*

    <type> <NomVar1>, <NomVar2>, ..., NomVarN> ;

  - Exemple en *C* :
    - int x, y ;
    - short compteur ;
    - float hauteur, largeur ;
    - double r ;
    - char touche ;

Chap. 3 : 2. Déclaration des variables simples

- **Les constantes littérales :**

  Dans un programme C, on peut manipuler les constantes littérales en nombre de 4 :

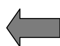
  - constantes entières,
  - constantes réelles,
  - constantes caractères et
  - constantes chaînes de caractères.

Chap. 3 : 2. Déclaration des variables simples 

- **2.1 Les constantes entières**
  - sous forme décimale : 100, 255.
  - sous forme octale, en faisant précéder le nombre par le caractère 0 (zéro) : 0144, 0377.
  - sous forme hexadécimale, en faisant précéder le nombre par 0x ou 0X : 0x64, 0Xff
  - **Remarques** :
    - Le type attribué à une constante est automatique (C choisit la solution la plus économique)
    - On peut forcer l'ordinateur à attribuer à la constante entière un type de notre choix, en employant l'un des suffixes suivants :

Suffixe	*Type*	*Exemple*
u ou **U**	**unsigned (int**	550u
l ou **L**	**long**	123456789L
ul ou **UL**	**unsigned long**	12092ul

Chap. 3 : 2. Déclaration des variables simples



- **2.2 Constantes réelles :**
  - en notation décimale : 123.4
  - en notation exponentielle : 1234e-1 ou bien 1234E-1
  - **Remarques :**
    - Par défaut, les constantes réelles sont du type double.
    - Le suffixe f ou F force l'utilisation du type float.
    - Le suffixe l ou L force l'utilisation du type long double.

Chap. 3 : 2. Déclaration des variables simples

- **2.3 Constantes caractères** :
  - Sont toujours indiqués entre apostrophes ' '
  - Exemples :
    - 'a' ; 'b' ; 'A' ; '+' ; ',' ; …
  - La valeur d'un caractère constant est son code ASCII. Les caractères constants peuvent donc apparaître dans des opérations arithmétiques ou logiques.
  - Ainsi, l'expression : 'a' + '?' vaut 160
    (le code ASCII de 'a' est égal à 97 alors que celui de '?' est 63)
  - Pour distinguer certains caractères spéciaux (les caractères de contrôle ou caractères non imprimables), on utilise le signe \ (antislash) → tableaux

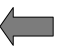
Chap. 3 : 2. Déclaration des variables simples



- **Séquences d'échappement** :

Caractère de contrôle	*Signification*
'\a'	Bip sonore
'\b'	Retour arrière (*back space*)
'\t'	Tabulation horizontale
'\n'	Passage à la ligne suivante
'\r'	Retour chariot
'\0'	Caractère nul
'\\'	Trait oblique (*antislash*)
'\?'	Point d'interrogation
'\''	Apostrophe
'\"'	Guillemets
'\f'	Saut de page
'\v'	Tabulation verticale

Chap. 3 : 2. Déclaration des variables simples



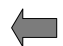
- **2.4 Constantes chaînes de caractères :**
  - Sont représentées entre guillemets " ".
  - **Exemple :** "Ceci est une chaîne de caractère"
  - Le **compilateur C** rajoute à la fin de toute chaîne de caractères le caractère nul '\0' pour indiquer sa fin.
  - Dans une chaîne de caractère, on peut utiliser les séquences d'échappement. L'instruction suivante :

```
printf("Bonjour, \n\tcomment vas-tu?\n")
```

produit la sortie :

```
Bonjour,
        comment vas-tu?
```

Chap. 3 : 2. Déclaration des variables simples



- **2.5 Initialisation des variables :**

  - En **C**, il est possible d'initialiser les variables à la déclaration.
  - **Exemples :**
    - **int** max = 1023 ;
    - **char** tabulation = '\t' ;

  - En utilisant l'attribut **const**, la valeur d'une variable ne change pas au cours du programme : C'est **une constante**.
  - **Exemples :**
    - **const int** MAX = 767 ;
    - **const char** NEWLINE = '\n' ;

Chap. 3 : 3. Opérateurs Standards

- Les opérateurs sont des symboles qui permettent de manipuler des variables, c'est-à-dire effectuer des opérations, les évaluer, ...

3.1 Opérateurs classiques

3.1.1 Opérateur d'affectation =

<variable> = <expression> ;

- L'expression est évaluée puis affectée à la variable.
- **Exemples :**

```
■ const int LONG = 141 ;      /* affectation de valeurs constantes */
■ char NATION = 'M' ;
■ int val, resultat ;
■ char lettre ;
  val = LONG ;                /* affectation de valeurs de variables */
  lettre = NATION ;
  resultat = 45 + 5 * val ;   /* affect. de valeurs d'expressions */
```

Chap. 3 : 3. Opérateurs Standards

- **3.1.2 Opérateurs arithmétiques**
  - +- * /
  - L'opérateur % permet d'obtenir le reste de la division entière.
  - L'opérateur / retourne un quotient entier si les deux opérandes sont entiers.

- **3.1.3 Opérateurs logiques**

&&	**:**	**ET logique (and)**
||	**:**	**OU logique (or)**
!	**:**	**négation logique (not)**

  - S'appliquent à des expressions booléennes (**0** si **faux** et **valeur non nulle** si **vrai**)
  - ET retourne la valeur **1** si les deux opérandes sont non nuls, et **0** sinon.
  - OU retourne la valeur **1** si au moins un des opérandes est non nul, et **0** sinon.

  - **Exemples**
    - L'expression : **32 && 40** vaut **1**
    - L'expression : !65.34 vaut **0**
    - L'expression : !!0 vaut **0**

Chap. 3 : 3. Opérateurs Standards

3.1.3 Opérateurs de comparaison

- = = : l'opérateur égal à.
- **!=** : l'opérateur différent de.
- **<, <=, >, >=** : plus petit, plus petit ou égal, plus grand, plus grand ou égal.
- Ces opérateurs retournent la valeur **0** si la comparaison est fausse et **1** sinon
- Exemple

0 || !(32 > 12) retourne la valeur 0.

Chap. 3 : 3. Opérateurs Standards

3.1.4 Opérateurs de bits

- Ils travaillent sur les **bits**.
- Les **opérandes** doivent être de type **entier** (char, short, int, long, signés ou non).
- **Opérateurs de décalage de bits**

>> : décalage à droite. **<< : décalage à gauche.**

  - L'opérande gauche constitue l'objet à décaler et l'opérande droit le nombre de bits de décalage.
  - Si la quantité à décaler est signée alors le bit signe (la position binaire d'extrême gauche) est préservée lors d'un décalage à droite, c.-à-d. ce bit se propage de façon à garder la signe de la donnée.
  - Si la quantité est non signée, alors les bits laissés libres sont mis à 0.

Chap. 3 : 3. Opérateurs Standards

3.1.4 Opérateurs de bits - Opérateurs de décalage de bits-

Exemples

- **int** i, j ;
- **unsigned int** u, v ;

- i = 5 ; /* *représentation* de i : 0000 0000 0000 0101 */
- j = i<<2 ; /* *représentation* de j : 0000 0000 0001 0100 c.-à-d. la valeur 20 */
- i = -3 ; /* *représ.* de i : 1111 1111 1111 1101 */
- j = i<<2 ; /* *représ.* de j : 1111 1111 1111 0100 c.-à-d. la valeur -12 */
- i = -32768 ; /* *représ.* de i : 1000 0000 0000 0000 */
- j = i>>2 ; /* *représ.* de j : 1110 0000 0000 0000 c.-à-d. la valeur -8192 */
- u = 32768 ; /* *représ.* de u : 1000 0000 0000 0000 */
- v = u>>2 ; /* *représ.* de v : 0010 0000 0000 0000 */

Chap. 3 : 3. Opérateurs Standards

3.1.4 Opérateurs de bits

Opérateurs bit à bit :

& : ET logique.

| : OU inclusif.

^ : OU exclusif.

~ : complément à 1.

- Ici, les opérateurs portent sur les bits de même rang.
- Rappel :

1&1 == 1	1|1 == 1	1^1 == 0
1&0 == 0	1|0 == 1	1^0 == 1
0&1 == 0	0|1 == 1	0^1 == 1
0&0 == 0	0|0 == 0	0^0 == 0

Chap. 3 : 3. Opérateurs Standards

3.1.4 Opérateurs de bits -Opérateurs bit à bit

Exemples

- unsigned int n, c ;
  n = … ;
  c = n & 0177 ; /* 0177 est égal à (82 + 7 * 8 + 7) */
  c est constitué des 7 bits de poids faible de n, complétés à gauche par des 0
- unsigned int n, c ;
  n = … ;
  c = n | 0177 ;
  c est constitué des bits de n, les 7 bits de poids faible étant forcés à 1
- unsigned int n, c ;
  n = … ;
  c = n ^ 0177 ;
  c est constitué des bits de n, les 7 bits de poids faible étant inversés

Chap. 3 : 3. Opérateurs Standards

3.2 Opérateurs particuliers de C

3.2.1 Opérateurs d'affectation

- Pour la plupart des expressions de la forme :

expr1 = (expr1) OP (expr2)

- Il existe une formulation équivalente utilisant un **opérateur d'affectation**:

expr1 OP= expr2

- Opérateurs d'affectation utilisables :

+=	-=	*=	/=	%=
<<=	>>=	&=	^=	|=

- **Exemples**

a = a + b s'écrit a += b

n = n << 2 s'écrit n <<= 2

Chap. 3 : 3. Opérateurs Standards

3.2 Opérateurs particuliers de C

3.2.2 Opérateurs d'incrémentation (++) et de décrémentation (--)

- **Post-incrémentation**

<var>++

- La valeur de la variable <var> est d'abord utilisée telle quelle, puis incrémentée.

- **Exemple**
  - int k, n ;
  - k = 0 ;

```
n = k++ ;    /* passe d'abord la valeur de k à n et incrémente après */
             /* ici, k vaut 1 et n vaut 0 */
```

Chap. 3 : 3. Opérateurs Standards

3.2 Opérateurs particuliers de C

3.2.2 Opérateurs d'incrémentation (++) et de décrémentation (--) :

- **Pré-incrémentation**

  ++<var>

  - **Exemple**

```
int k, n ;
k = 0 ;
n = ++k ;      /* incrémente d'abord et passe la valeur incrémentée à n */
               /* ici, k vaut 1 et n vaut 1 */
```

- **Post-décrémentation**   **<var>--**
- **Pré-décrémentation**   **--<var>**

Chap. 3 : 3. Opérateurs Standards

3.2 Opérateurs particuliers de C

3.2.3 Opérateur séquentiel (,)

<expr1> , <expr2>,…, <exprN>

- Exprime des calculs successifs dans une même expression
- Le type et la valeur de l'expression sont ceux du dernier opérande.
- **Exemple** :

  L'expression : **x = 5 , x + 6** a pour valeur 11

3.2.4 Opérateur conditionnel (? :)

<expression> ? <expr1> : <expr2>

- <expression> est évaluée. Si sa valeur est non nulle, alors la valeur de <expr1> est retournée. Sinon, c'est la valeur de <expr2> qui est renvoyée.
- **Exemple**

  max = a > b ? a : b

  si a est le plus grand, alors affectation à max du contenu de a sinon affectation du contenu de b

Chap. 3 : 3. Opérateurs Standards



3.2 Opérateurs particuliers de C

- **3.2.3 Opérateurs sizeof**

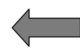
sizeof(<type>) ou **sizeof**(<variable>)

  - Retourne le nombre d'octets occupés par le type de données ou la variable spécifiés.
  - Exemple
    - **int** x ;
    - **sizeof**(**int**) /* retourne la valeur 2 (le type **int** occupe 2 octets). */
    - **sizeof**(x) /* retourne la valeur 2. */

Chap. 3 : 4. Expressions et Instructions

- Une **expression** est un **calcul** qui donne une **valeur comme résultat** (exemple : 8+5).
- Une **expression** peut comporter des **variables** et des **constantes** combinés entre eux par des **opérateurs** et former ainsi une **expression complexe**
- Les **expressions** peuvent contenir des **appels de fonctions** et elles peuvent **apparaître** comme **paramètres** dans des appels de **fonctions.**
- Toute **expression** suivie d'un **point virgule** devient une **instruction**.
- Exemples
  - i = 0;
  - i++;
  - a = (5 * x + 100 * y) * 2;
  - x = **pow**(a, 4); /* fonction puissance : **pow**(x, y) ↔ x^y */

Chap. 3 : 4. Expressions et Instructions



Remarque

- Les **affectations** sont aussi **interprétées** comme des **expressions**. **L'opérateur d'affectation** retourne la **valeur affectée**.

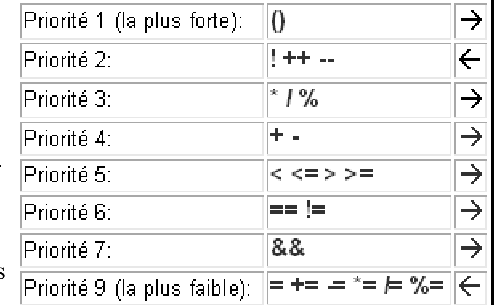
- On peut **enchaîner des affectations**. L'**évaluation** commence de la **droite vers la gauche**.

- **Exemples**
  - **b=(a = 5 + 3)+1** a = **8 et b = 9**
  - **a = b = c = d** équivalente à : **a = (b = (c = d))**



Chap. 3 : 5. Priorité et associativité des opérateurs

- Lors de l'évaluation des différentes parties d'une expression, les opérateurs respectent certaines lois de priorité et d'associativité.
- **Exemples**
  - La multiplication a la priorité sur l'addition
  - La multiplication et l'addition ont la priorité sur l'affectation.
- **Tableau des opérateurs et priorité**
  - **La priorité** est décroissante de haut en bas dans le tableau.
  - **La règle d'associativité** s'applique pour tous les opérateurs d'un même niveau de priorité. (→ pour une associativité de gauche à droite et ← pour une associativité de droite à gauche).
  - Dans une expression, les parenthèses forcent la priorité.



Priorité 1 (la plus forte):	()	→
Priorité 2:	! ++ --	←
Priorité 3:	* / %	→
Priorité 4:	+ -	→
Priorité 5:	< <= > >=	→
Priorité 6:	== !=	→
Priorité 7:	&&	→
Priorité 9 (la plus faible):	= += -= *= /= %=	←

Chap. 3 : 5. Priorité et associativité des opérateurs

- **Exemples**

  - **a = b = c** s'interprète comme **a = (b = c)**

  - a *= b += 5 s'évalue (a = 3 ; b = 4 ;) :
    b += 5 → 9
    a *= 9 → a = 27

  - !--a == ++!b s'évalue (a = 1 ; b = 4 ;) :
    !b → 0
    ++0 → 1
    --a → 0
    !0 → 1
    1 == 1 → 1

Chap. 3 : 6. Conversions de type (cast)

6.1 Conversion automatique

- Si un opérateur a des opérandes de différents types, les valeurs des opérandes sont converties automatiquement dans un type commun.

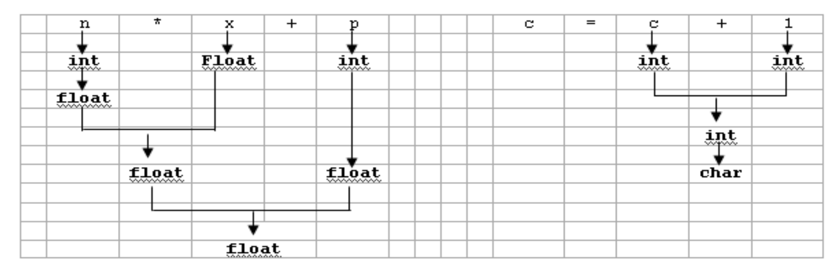
- **Règle de conversion automatique**
  Lors d'une opération avec :
  - deux entiers : les types **char** et **short** sont convertis en **int**. Ensuite, il est choisit le plus large des deux types dans l'échelle : **int, unsigned int, long, unsigned long.**
  - un entier et un réel : le type entier est converti dans le type du **réel**.
  - deux réels : il est choisit le **plus large** des deux types selon l'échelle : **float, double, long double**.
  - Dans une affectation : le **résultat** est toujours converti dans le type de la destination. **Si ce type est plus faible, il peut y avoir une perte de précision.**

Chap. 3 : 6. Conversions de type (cast)

6.1 Conversion automatique

Exemples :

1.
```
int n, p ;
float x ;
char c ;
```
Donner le type de chacune des expressions suivantes :
```
n * x + p
c = c + 1
```



2. *Exemple de perte de précision* :
```
unsigned int a = 70 000 ; /* la valeur de a sera 4464 */
/* ceci est accepté, aucun avertissement du compilateur C */
```


Chap. 3 : 6. Conversions de type



6.2 Conversion forcées(casting)

- Le type d'une expression peut être forcé, en utilisant **l'opérateur cast** :

(<type>) <expression>

- **Exemple**
  - **char** a = 49; // **a = '1'**
  - **int** b = 4 ;
  - **float** c ;

c = (float) a / b;

  - Quelle est la valeur de c ?

12.25

Chap. 4 : Lire & Ecrire des données

4.1 Ecriture formatée de données : printf()

- La fonction printf permet d'afficher du texte, des valeurs de variables ou des résultats d'expressions sur écran (sortie standard).

- **Forme générale** :

  printf("<format>", <expr1>, <expr2>, …, <exprN>) ;

- La partie **"<format>"** est une chaîne de caractères qui peut contenir :
  - du **texte**
  - des **caractères de contrôle** ('\n' , '\t' , …)
  - des **spécificateurs de format**.

Chap. 4 : Lire & Ecrire des données

4.1 Ecriture formatée de données : printf()

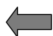
- La partie **"<format>"** contient exactement **un spécificateur** pour **chaque expression** <expr1>, <expr2>, … et <exprN>.

- Les **spécificateurs de format** commencent toujours par le symbole **%** et se terminent par **un ou deux caractères** qui indiquent le **format d'affichage**.

Spécificateurs de format pour printf:

Spécificateur	*Rôle (afficher :)*	*Type*
%c	un seul caractère	**char**
%d ou %i	un entier relatif	**int**
%u	un entier naturel (non signé)	**unsigned int**
%o	un entier sous forme octale	**int**
%x	un entier sous forme hexadécimale (a-f)	**int**
%X	un entier sous forme hexadécimale (A-F)	**int**
%f	un réel	**float** ou **double**
%e	un réel en notation exponentielle (e)	**float** ou **double**
%E	un réel en notation exponentielle (E)	**float** ou **double**
%s	une chaîne de caractères	**char***

Chap. 4 : Lire & Ecrire des données



4.1 Ecriture formatée de données : printf()

- **Exemples :**
  - La suite d'instructions :
    - int a = 1234 ;
    - int b = 566 ;

      printf("**%i** plus **%i** est **%i**\n", a, b, a + b) ;
    - va afficher sur l'écran :

      1234 plus 566 est 1800
  - La suite d'instructions :
    - char b = 'A' ; /* le code ASCII de A est 65 */

      printf("Le caractère **%c** a le code **%i**\n", b, b) ;
    - va afficher sur l'écran :

      Le caractère A a le code 65

Chap. 4 : Lire & Ecrire des données

4.2 Lecture formatée de données : scanf()

- **scanf** lit depuis le clavier (entrée standard). Elle fait correspondre les caractères lus au format indiqué dans la chaîne de format.
- La **spécification de formats** pour scanf est identique à celle de printf, sauf qu'au lieu de fournir comme arguments des variables à scanf, ce sont **les adresses de ces variables que l'on transmet.**
- L'**adresse d'une variable** est indiquée par le **nom de la variable** précédé du signe **&.**
- **Forme générale** :

scanf("<format>", <AdrVar1>, <AdrVar2>, …, <AdrVarN>)

Chap. 4 : Lire & Ecrire des données

4.2 Lecture formatée de données : scantf()

- **Exemples** :
  - int jour, mois, annee ;
    scanf("**%i %i %i**", &jour, &mois, &annee) ;
  - Cette instruction lit 3 entiers séparés par les espaces, tabulations ou interlignes. Les valeurs sont attribuées respectivement aux 3 variables : jour, mois et annee.
  - int i ;
  - float x ;
    scanf("**%d %f**", &i, &x) ;
  - Si lors de l'exécution, on entre 48 et 38.3e-1 alors scanf affecte 48 à i et 38.3e-1 à x.

Chap. 4 : Lire & Ecrire des données



4.2 Lecture formatée de données : scanf()

- **Remarques** :
  - Lors de l'évaluation des données, **scanf** s'arrête si la chaîne de format a été travaillée jusqu'à la fin ou si une donnée ne correspond pas au format indiqué.
  - **scanf** retourne comme résultat le nombre d'arguments correctement reçus et affectés.
  - **Exemples** :
    - int jour, mois, annee, recu ;
    - recu = scanf("**%i %i %i**", &jour, &mois, &annee) ;

Données introduites	*Affichage à l'écran*			
	recu	*jour*	*mois*	*annee*
12 4 1980	3	12	4	1980
12/4/1980	1	12	indéfinie	indéfinie
12.4 1980	1	12	indéfinie	indéfinie
12 4 19.80	3	12	4	19

Chap. 4 : Lire & Ecrire des données

4.2 Ecriture d'un caractère : putchar()

- **putchar** permet d'afficher un caractère sur l'écran.
- **putchar(c)** ; est équivalente à **printf("%c", c)** ;
- **Forme générale** :

  putchar(<caractere>) ;

- Elle reçoit comme argument la valeur d'un caractère convertie en entier.

Chap. 4 : Lire & Ecrire des données



4.2 Ecriture d'un caractère : putchar()

Exemples

- char a = 63 ;
- char b = '\n' ;
- int c = '\a' ;

```
putchar('x') ; /* affiche la lettre x */
putchar('?') ; /* affiche le symbole ? */
putchar(b) ;  /* retour à la ligne */
putchar(65) ; /* affiche le caractère de code ASCII = 65  c.-à-d. la lettre A */
putchar(a) ;  /* affiche le caractère de code ASCII = 63  c.-à-d. le symbole ? */
putchar(c) ;  /* beep sonore */
```

- **Remarque :**

putchar retourne la **valeur du caractère** écrit toujours considéré comme un entier, ou bien la **valeur -1 (EOF)** en cas d'erreur.

Chap. 4 : Lire & Ecrire des données

4.2 Lecture d'un caractère : getchar()

- Permet de lire un **caractère** depuis le **clavier**.
- **c=getchar(c)** ; est équivalente à **scanf("%c",&c)** ;
- *Forme générale* :

  <Caractere> = getchar() ;

- **Remarques** :
  - **getchar** retourne le caractère lu (un entier entre 0 et 255), ou bien la valeur -1 (EOF).
  - **getchar** lit les données depuis le clavier et fournit les données après confirmation par la touche **"entrée"**
- **Exemple** :
  - int c ;

```
c = getchar() ;    /* attend la saisie d'un caractère au clavier */
```


Chap. 5 : **Structures de contrôle**

- On appelle structure de contrôle toute instruction qui permet de contrôler le fonctionnement d'un programme.
- Parmi les structures de contrôle, on distingue :
  - structures de choix et
  - structures répétitives

Chap. 5 : Structures de contrôle

5.1 Structures de choix

- Les structures de choix permettent de déterminer quelles instructions seront exécutées et dans quel ordre.
- En langage C, les structures de choix peuvent être exprimées par :
  - L'instruction de branchement conditionnels : **if…else**
  - l'instruction de branchement multiple : **switch**

Chap. 5 : Structures de contrôle

5.1 Structures de choix

5.1.1 Branchement conditionnel (if … else)

Format :

```
if (expression) bloc-instruction-1
else  bloc-instruction-2
```

où

- expression : est une expression quelconque. Après évaluation, si elle est vraie, alors le 1er bloc d'instructions est exécuté, sinon c'est le 2ème bloc qui est exécuté.
- bloc d'instructions : peut désigner **une suite d'instructions délimitées par des accolades** ou une seule instruction.

Chap. 5 : Structures de contrôle

5.1 Structures de choix

5.1.1 Branchement conditionnel (if … else)

Remarques :

- On notera que l'expression conditionnelle doit être entre parenthèses.
- La partie else est optionnelle :
  if (expression) bloc-instruction
- Lorsque plusieurs instructions if sont imbriquées, il est convenu que chaque else se rapporte au *dernier* if qui ne possède pas de partie else.

Chap. 5 : Structures de contrôle



5.1 Structures de choix

5.1.1 Branchement conditionnel (if … else)

Exemples :

- Calcul du maximum de deux entiers
- Teste si un caractère saisie depuis le clavier , est une lettre majuscule , si oui il affiche cette lettre en minuscule, sinon il affiche un message d'erreur.
- Résolution de l'équation du second degré : $a.x^2 + b.x + c = 0$

Chap. 5 : Structures de contrôle



5.1 Structures de choix

5.1.2 Branchement multiple (switch)

On l'appelle aussi l'*instruction d'aiguillage. Elle t*este si une expression prend une valeur parmi *une suite de constantes*, et effectue le branchement correspondant si c'est le cas.

Format :

```
switch ( expression )
{
  case  constante1 :  suite d'instructions 1
  case  constante2 :  suite d'instructions 2
  ...
  case  constanteN : suite d'instructions N
  default :  suite d'instructions
}
```


Chap. 5 : Structures de contrôle

5.1 Structures de choix

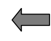
5.1.2 Branchement multiple (switch)

Remarques **:**

- Le fonctionnement de cette instruction est le suivant :
  - expression est évaluée ;
  - s'il existe un énoncé case avec une constante qui égale la valeur de expression, le contrôle est transféré à l'instruction qui suit cet énoncé;
  - si un tel case n'existe pas, et si énoncé default existe, alors le contrôle est transféré à l'instruction qui suit l'énoncé default ;
  - si la valeur de expression ne correspond à aucun énoncé case et s'il n'y a pas d'énoncé default, alors aucune instruction n'est exécutée.

- **Attention**. Lorsqu'il y a branchement réussi à un case, toutes les instructions qui le suivent sont exécutées, jusqu'à la fin du bloc ou jusqu'à une instruction de rupture (break).

Chap. 5 : Structures de contrôle



5.1 Structures de choix 5.1.2 Branchement multiple (switch)

Exemples :

```
#include <stdio.h>
main()
{  int a,b ,
   char operateur ;
      printf("Entrez un opérateur (+, -, * ou /) : ") ;
    scanf("%c", &operateur) ;
    printf("Entrez deux entiers : ") ;
    scanf("%d  %d", &a,&b) ;
   switch (operateur)
   {
      case '+' : printf("a + b = %d\n",a+b) ; break ;
      case '-'  : printf("a - b = %d\n",a-b) ; break ;
      case '*' : printf(" a * b = %d\n",a*b) ; break ;
      case '/' : printf("a / b = %d\n",a/b) ; break ;
     default  : printf("opérateur inconnu\n") ; break ;
   }
   return 0 ;
}
```

Soit le code C suivant :

```
int i , j;
... //initialisation de i
j = 0;
switch (i)
{
    case 3: j++;
    case 2: j++;
    case 1: j++;
}
```

- Si on suppose que i ne peut prendre que les valeurs 0 ... 3.
- Que fait ce programme?

Chap. 5 : Structures de contrôle

5.2 Structures répétitives (Boucles)

- Les structures répétitives (ou Boucles) permettent de répéter une série d'instructions tant qu'une certaine condition reste vraie.
- On appelle parfois ces structures instructions d'itérations.
- En langage C, les structures répétitives peuvent être exprimées par :
  - les instructions while et do ... while
  - l'instruction for

Chap. 5 : Structures de contrôle

5.2.1 while et do … while

- Les instructions **while** et **do … while** représentent un moyen d'exécuter plusieurs fois la même série d'instructions.
- La syntaxe :

```
while ( condition )
{
        liste d'instructions
}
```

```
do
{
        liste d'instructions
}
while ( condition );
```

- **Dans la structure while on vérifie la condition avant d’exécuter la liste d’instructions, tandis que dans la structure do … while on exécute la liste d’instructions avant de vérifier la condition**

Chap. 5 : Structures de contrôle



5.2.1 while et do … while

Exemples

1. Programme pour imprimer les entiers de 1 à 9.

```
i = 1;

while (i < 10)
{
    printf("\n i = %d",i);
    i++;
}
```

2. Programme pour saisir au clavier un entier entre 1 et 10 :

```
int a;

do
{
  printf("\n Entrez un entier entre 1 et 10 : ");
  scanf("%d",&a);
}
while ((a <= 0) || (a > 10));
```

Chap. 5 : Structures de contrôle

5.2.1 for

- A l'instar des instructions **while** et **do … while, l'instruction for permet** d'exécuter plusieurs fois la même série d'instructions.
- La syntaxe de for est :

```
for ( expression1 ; expression2 ; expression3 )
{
        liste d'instructions
}
```

- Dans la construction de for :
  - **expression1** : effectue les initialisations nécessaires avant l'entrée dans la boucle;
  - **expression2** : est le test de continuation de la boucle ; le test est évalué avant l'exécution du corps de la boucle;
  - **expression3** : est évaluée à la fin du corps de la boucle.

Chap. 5 : Structures de contrôle

5.2.1 for

Remarques

- En pratique, expression1 et expression3 contiennent souvent plusieurs initialisations séparées par des virgules.
- Les expressions *expression*1 et *expression*3 peuvent être absentes (les points-virgules doivent cependant apparaître). **Par exemple** : for (; *expression*2 ;)
- Lorsque expression2 est absente, l'expression correspondante est considérée comme vraie. Par conséquent, **for** (; ;) est une boucle infinie.

Chap. 5 : Structures de contrôle

5.2.1 for

Remarques(suite)

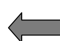
- Par définition, la construction de for

```
for ( expression1 ; expression2 ; expression3 )
{
        liste d'instructions
}
```

est équivaut *strictement à* celle-ci :

```
expression1;
while (expression2)
{
    liste instructions;
    expression3;
}
```


Chap. 5 : Structures de contrôle



5.2.1 for

Exemples

1. Programme pour calculer la somme de 1 à 100 :

```
int n, total ;

for ( total = 0, n = 1 ; n<101 ; n++ )
        total += n ;

printf("La somme des nombres de 1 à 100 est %d\n", total) ;
```

2. Programme pour calculer la factorielle d'un entier n :

```
int n , i , fact ;

for ( i = 1, fact = 1 ; i <= n ; i++)
        fact *= i;

printf("%d ! = %d \n",n,fact);
```

Chap. 5 : Structures de contrôle

5.3 Instructions break et continue

5.3.1 l’instruction break

- On a vu le rôle de l'instruction break; au sein d'une instruction de branchement multiple switch.
- L'instruction break peut, plus généralement, être employée à l'intérieur de n'importe quelle boucle (for ; while ; do …while ; switch). Elle permet l’abandon de la structure et le passage à la première instruction qui suit la structure.
- En cas de boucles imbriquées, break fait sortir de la boucle la plus interne.

Chap. 5 : Structures de contrôle

5.3 Instructions break et continue 5.3.1 l’instruction break

Exemples (break)

1. Que fait ce programme?

```
for ( ; ; )
{
    printf("donne un nombre (0 pour sortir) : ");
    scanf("%d", &n);
   if (n == 0) break;
   ...
   exploitation de la donnée n
   ...
}
```

2. Que fait ce programme?

```
#include <stdio.h>
int main( )
{
  int i, j ;

  for (i = 1 ; i<=15 ; i++)
  {
    for (j = 1 ; j<=15 ; j++)
     {
       if (j = = 5)   break ;
       printf("%d\t", i * j) ;
     }
    printf("\n") ;
  }
  return 0 ;
}
```

Chap. 5 : Structures de contrôle

5.3 Instructions break et continue

5.3.1 l'instruction continue

- L'instruction continue peut être employée à l'intérieur d'une structure de type boucle (for ; while ; do …while).
- Elle produit l'abandon de l'itération courante et fait passer directement à l'itération suivante d'une boucle
- L'instruction continue concerne la boucle la plus proche.

Chap. 5 : Structures de contrôle

5.3 Instructions break et continue **5.3.2 l'instruction continue**

Exemple (continue)

1. Que fait ce programme?

```
int main()
{
    int i, j ;
    ... //initialisation de i et j
    for ( ; i>0 && j>0 ; i--, j-- )
   {
       if ( i = = 5 )     continue ;
       printf("i : %d et j : %d\n, i, j) ;
       if ( j = = 5 )
       break ;
    }
    return 0 ;
}
```

Valeurs introduites	Affichage
i = 2 et j = 3	i : 2 et j : 3 i : 1 et j : 2
i = 6 et j = 3	i : 6 et j : 3 i : 4 et j : 1
i = 3 et j = 5	i : 3 et j : 5

Université Mohammed V-Agdal
Faculté des sciences de Rabat
Département de Mathématique et Informatique

Langage C

Préparé et présenté par
Pr. M. Benchrifa

Année Universitaire 2006/2007

- ***CHAPITRE 6*** **: Tableaux**
  - Définition
  - Tableaux à une dimension (Vecteurs)
    - Déclaration ; Mémorisation ; …
  - Tableaux à plusieurs dimensions
    - Déclaration
    - Tableaux à deux dimensions (matrices) :
      - *Déclaration ; Mémorisation ; …*

Chap. 6 : Tableaux - Introduction

- Les variables, telles que nous les avons vues, ne permettent de stocker qu'une seule donnée à la fois.
- Pour mémoriser et manipuler de nombreuses données (100, 1000, …), des variables distinctes seraient beaucoup trop lourdes à gérer.
- Pour résoudre ce problème, le langage C (ainsi que les autres langages de programmations) propose une structure de données permettant de stocker l'ensemble de ces données dans une "variable commune" appelée :

Tableau

Chap. 6 : Tableaux

1. Définition

- On appelle tableau une variable composée de données de même type, stockée de manière contiguë en mémoire (les unes à la suite des autres).
- Un tableau est une suite des éléments de même taille. Par conséquent, la taille (en octet) du tableau est :

  taille (en octet) du type de donnée * le nombre d'élément

- Le type des éléments du tableau peut être :
  - simple : char, int, float, double, ...
    → tableau à une dimension ou tableau unidimensionnel
  - tableau
    → tableau à plusieurs dimensions ou tableau multidimensionnel
  - Autres : Pointeurs (clef chapitre 7) et Structures (clef chapitre 8)

Chap. 6 : Tableaux

2. Tableaux à une dimension

2.1 Déclaration

La déclaration d'un tableau à une dimension se fait de la façon suivante :

<Type Simple> Nom_du_Tableau [Nombre_Elements];

Type Simple : définit le type d'élément que contient le tableau
Nom_du_Tableau : est le nom que l'on décide de donner au tableau, le nom du tableau suit les mêmes règles qu'un nom de variable.
Nombre_Elements : est une expression constante entière positive.

Exemples :

```
char      caractere[12] ;      //Taille en octet : 1 octet  * 12 = 12 octets
int       entier[10] ;         //Taille en octet : 2 octets * 10 = 20 octets
float     reel_SP[8] ;         //Taille en octet : 4 octets * 8 = 32 octets
double    reel_DP[15] ;        //Taille en octet : 8 octets * 15 = 120 octets
```

Chap. 6 : Tableaux 2. Tableaux à une dimension

2.3 Initialisation à la déclaration

Il est possible d'initialiser le tableau à la définition :

<Type> Tableau [Nombre_Elements] = {C1, C2, … , Cn};

Où C1, C2, .., Cn sont des constantes dont le nombre ne doit pas dépasser le Nombre_Elements.

Si la liste de constantes ne contient pas assez de valeurs pour tous les éléments, les éléments restantes sont initialisées à zéro.

Exemples :

```
char   voyelles[6]           = { 'a' , 'e' , 'i' , 'o' , 'u' , 'y' } ;
int    Tableau_entier1[10]   = {10 , 5 , 9 , -2 , 011 , 0xaf , 0XBDE};
float  Tableau_reel[8]       = { 1.5 , 1.5e3 ,  0.7E4 };
Short A[3]                   = {12 , 23 , 34 , 45 , 56};            //Erreur !
int Tableau_entier2[] = { 15 , -8 , 027 , 0XABDE }   //Réservation automatique
                                                     //Tableau de 4 éléments
```

(The line "Short A[3] = {12 , 23 , 34 , 45 , 56};" is struck through on the page.)

Chap. 6 : Tableaux **2. Tableaux à une dimension (Vecteurs)**

2.4 Accès aux composantes d'un tableau

Pour accéder à un élément du tableau, il suffit de donner le nom du tableau, suivi de l'indice de l'élément entre crochets :

Nom_du_Tableau [indice]

Où indice est une expression entière positive ou nulle.

Un indice est toujours positif ou nul ;
L'indice du premier élément du tableau est 0 ;
L'indice du dernier élément du tableau est égal au nombre d'éléments – 1.

Exemples :

```
short A[5]  = {12 , 23 , 34 , 45 , 56}; // A[0] = 12 ;  A[1] = 23 ; A[2] = 34 ;
                                        //  A[3] = 45 ;  A[4] = 56 ;

for( i = 0 ;  i< 5 ; i++ )                //Affichage des éléments
  printf(''A[%d] = %d \t'',i,A[i]);       // du tableau A
```

Chap. 6 : Tableaux **2. Tableaux à une dimension (Vecteurs)**

2.5 Remarques

- Chaque élément (TAB[i]) d'un tableau (int TAB[20]) est manipulé comme une simple variable, on peut :
  - la lire : scanf("%d", &TAB[i]); // TAB[i] sera initialisé par un entier saisi //depuis la clavier
  - l'écrire : printf("TAB[%d] = %d", i , TAB[i]); //Le contenu de TAB[i] //sera affiché sur écran
  - la passer en argument à une fonction : TAB[i] = pow(TAB[i],2); // = TAB[i] ^{2}
- Pour initialiser un tableau (TAB1) par les éléments d'un autre tableau (TAB2) :
  - évitez d'écrire **TAB1 = TAB2** (**<u>incorrect</u>**)
  - On peut par exemple écrire :

```
for(  i = 0  ;  i < taille_tableau  ;  i++ )
      TAB1[i] = TAB2[i];
```

Chap. 6 : Tableaux 2. Tableaux à une dimension 2.1 Déclaration

2.6 Déclaration et représentation en mémoire d'un tableau:

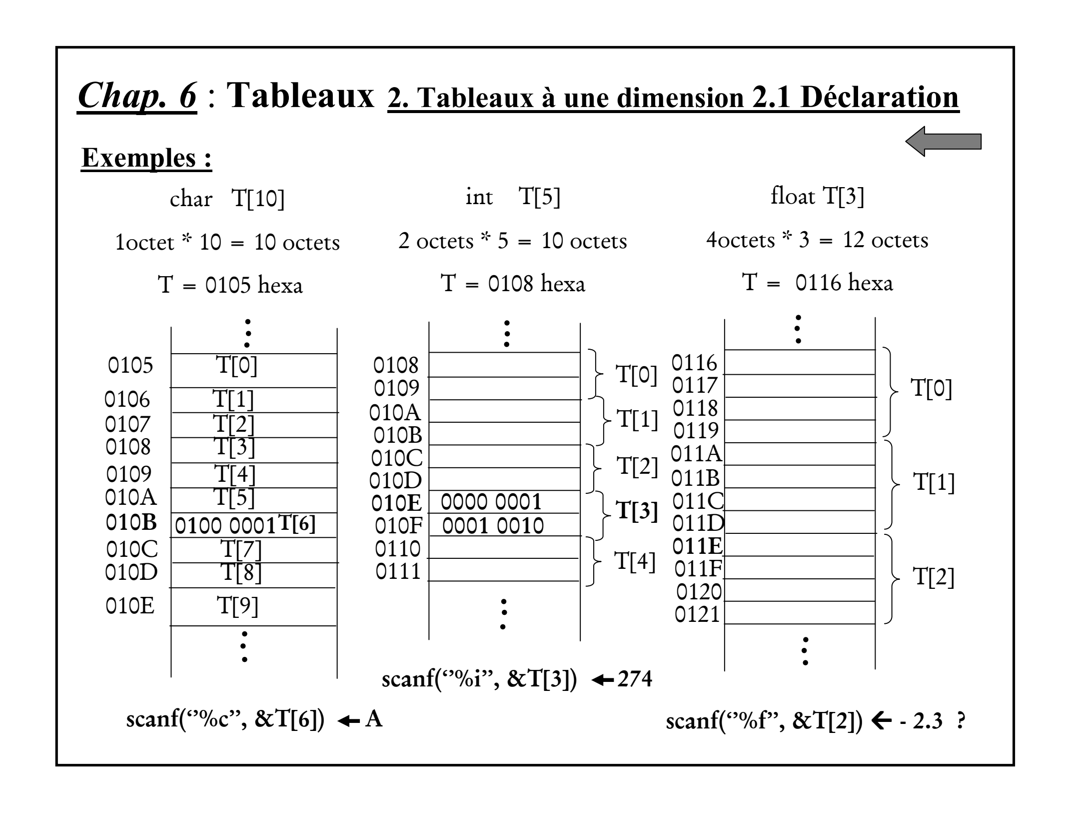
- La déclaration d'une variable au sein d'un programme provoque la réservation automatique, par le compilateur, d'un espace de la mémoire. Ce qui permettra, par conséquent, de conserver les valeurs assignées à cette variable le long de l'exécution du programme.
- Rappelons que la mémoire (RAM) est une superposition de cases mémoires ou rangées.
- Chaque case mémoire est une suite de 8 bits (1 octets). La lecture ou l'écriture en mémoire se fait par octet ou par mot machine (2 octets ou 4 octets).
- Chaque case mémoire est identifiée par un numéro appelé **adresse**. Par convention, la première case mémoire est identifiée par l'adresse 0, la seconde par adresse 1, …, jusqu'à 2^n-1 avec n le nombre de bits pour l'adressage.
- Souvent on est ramené à exprimer cette adresse en hexadécimal. Une écriture plus compact proche de la représentation binaire de l'adresse et donne une idée sur le nombre des bits d'adressage. Dans le cas ou n = 16, on dit un adressage sur 16 bits. Dans des machines, on peut avoir un adressage sur n = 32 bits ou n = 64 bits.

Chap. 6 : Tableaux 2. Tableaux à une dimension 2.1 Déclaration

2.2 Déclaration et représentation en mémoire d'un tableau (suite):

- En C, la déclaration d'un tableau T induit une réservation automatique d'une zone mémoire contiguë (les cases mémoire sont successives).
- Dés le déclaration, on connaît La taille d'un tableau T qui est conditionnée par le nombre N des éléments et leur type :
  Taille de T en octet = N * sizeof (type)
- En C, le nom d'un tableau **T** (l'identificateur) est un pointeur constant qui pointe sur le premier élément du tableau. Il contient l'adresse du premier élément du tableau.
- Etant donnée l'adresse du premier élément du tableau (matérialisée par le nom du tableau T), l'adresse du ième élément de T est donnée par :

&T[i] = &T[0] + sizeof(type) * i ou
&T[i] = T + sizeof(type) * i



Chap. 6 : Tableaux 2. Tableaux à une dimension (Vecteurs)



2.5 Exemples

- Saisie et affichage des données d'un tableaux d'entiers de 20 éléments aux maximum.
- Déterminer la plus petite valeur du tableau d'entiers A. afficher ensuite la valeur et la position du minimum. Si le tableau contient plusieurs minimum, retenir la position du premier minimum rencontré.
- Recherche d'une valeur dans un tableau **:** Etant donnés un tableau et une valeur. On recherche cette valeur dans le tableau. S'il existe, on affiche sa position sinon on affiche un message d'erreur

Chap. 6 : Tableaux

3. Tableaux à plusieurs dimensions

3.1 Déclaration

De manière similaire, on peut déclarer un tableau à plusieurs dimensions :

<Type Simple> Nom_du_Tableau [Nbre_E_1] [Nbre_E_2]…[Nbre_E_N];

- **Chaque élément entre crochets désigne le nombre d'éléments dans chaque dimension ;**
- **Le nombre de dimension n'est pas limité.**

3.2 Tableaux à deux dimensions (matrices)

3.2.1Déclaration

<Type Simple> Nom_du_Tableau [Nombre_ligne] [Nombre_colonne];

Exemple :

int T[3][4] ; //Taille en octet : 2 octets * 3 * 4 = 24 octets

On peut représenter un tel tableau de la manière suivante :

T[0][0]	T[0][1]	T[0][2]	T[0][3]
T[1][0]	T[1][1]	T[1][2]	T[1][3]
T[2][0]	T[2][1]	T[2][2]	T[2][3]

Chap. 6 : Tableaux 3.2 Tableaux à deux dimensions (matrices)

3.2.2 Initialisation à la déclaration et accès aux éléments :

Les valeurs sont affectées ligne par ligne lors de l'initialisation à la déclaration;

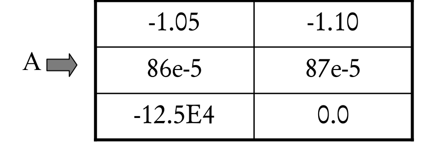
Accès aux composantes se fait par : Nom_tableau[ligne][colonne].

Exemples :

float A[3][2] = { {-1.05,-1.10} , {86e-5, 87e-5} , {-12.5E4} };

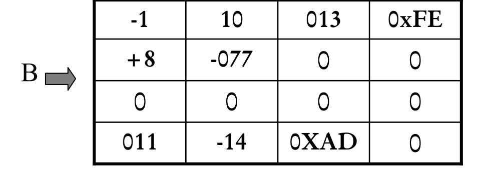
int B[4][4] = { {-1 , 10 , 013 , Oxfe} , {+8 , -077} , {} , {011,-14,0XAD} };

A ➡



-1.05	-1.10
86e-5	87e-5
-12.5E4	0.0

B ➡



-1	10	013	0xFE
+8	-077	0	0
0	0	0	0
011	-14	0XAD	0

Chap. 6 : Tableaux 3.2 Tableaux à deux dimensions (matrices)

3.2.3 **Déclaration et représentation en mémoire d'un tableaux à 2 dimensions :**

La déclaration d'un tableau T à deux dimensions (matrice) induit la réservation de l'espace mémoire nécessaire pour accueillir tous les éléments du tableau.
Les éléments de la matrice sont répartis en mémoires ligne par ligne.

Par exemple, soit T un tableau défini par : char T[3][4]

Les écritures suivantes sont équivalentes :

T[0] ≡ &T[0][0] : adresse du 1er élément

T[1] ≡ &T[1][0] : adresse du 1er élément de la 2ème ligne

T[2] ≡ &T[2][0] : adresse du 1er élément de la 3ème ligne

T[i] ≡ &T[i][0] : adresse du 1er élément de la ième ligne

Répartition des éléments de T en mémoire

	Mémoire
	:
T[0] →	T[0][0]
	T[0][1]
	T[0][2]
	T[0][3]
T[1] →	T[1][0]
	T[1][1]
	T[1][2]
	T[1][3]
T[2] →	T[2][0]
	T[2][1]
	T[2][2]
	T[2][3]
	:

Chap. 6 : Tableaux 3.2 Tableaux à deux dimensions (matrices)

3.2.3 Exemples

- Saisie et affichage des données entières d'une matrice de M ligne et N colonnes. 20 éléments aux maximum. M et N sont entrées au clavier.

- Produit de deux matrices : En multipliant une matrice A de M lignes et N colonnes avec une matrice B de N lignes et P colonnes, on obtient une matrice C de M lignes et P colonnes :
  La composante cij de la matrice C, placée à la ième ligne et jème colonne, se calcule de la façon suivante :

$$c_{ij} = \sum_{k=1}^{M} a_{ik}\, b_{kj} \qquad \text{où } 1 \le i \le N \text{ et } 1 \le j \le P$$

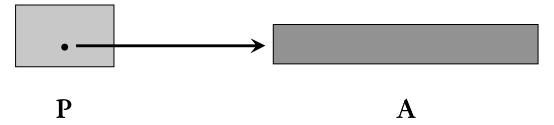
- *CHAPITRE 7* : **Pointeurs en langage C**
  - Introduction : Définition et Intérêts
  - Déclaration et initialisation d'un pointeur
  - Opérations élémentaires sur les pointeurs
  - Pointeurs et tableaux
  - Tableaux de pointeurs
  - Allocation dynamique de la mémoire

CHAPITRE 7 : Pointeurs en langage C

1. Introduction

1.1 Définition

- Un **pointeur** est une variable spéciale qui peut contenir l'**adresse** d'une autre variable.
- En C, chaque pointeur est limité à un type de données. Il peut contenir l'adresse d'une variable de ce type.
- Si un pointeur P contient l'adresse d'une variable A, on dit que **'P pointe sur A'**.



CHAPITRE 7 : Pointeurs en langage C

1. Introduction

1.2 Intérêts

- En C, l'utilisation de pointeurs est incontournable car ils sont étroitement liés à la représentation des tableaux
- Les **principales intérêts** des pointeurs résident dans la possibilité de :
  - Allouer de la mémoire dynamique sur le TAS, ce qui permet la gestion de structures de taille variable. Par exemple, tableau de taille variable.
  - Permettre le passage par référence pour des paramètres des fonctions (clef chapitre 8)
  - Réaliser des structures de données récursives (listes et arbres) (clef cours I4 structures de données)
  - …

CHAPITRE 7 : Pointeurs en langage C

2. Déclaration et initialisation d'un pointeur

2.1 Déclaration

- Un pointeur est une variable dont la valeur est égale a l'adresse d'une autre variable. En C, on **déclare un pointeur** par l'instruction :

type *nom_du_pointeur ;

où

- type est le type de la variable pointée,
- l'identificateur nom_du_pointeur est le nom de la variable pointeur et
- * est l'opérateur qui indiquera au compilateur que c'est un pointeur.

- Exemple : **int *p;**

  On dira que :

  p est un pointeur sur une variable du type **int** , ou bien

  p peut contenir l'adresse d'une variable du type **int**

  *p est de type int, c'est l'emplacement mémoire pointé par p.

CHAPITRE 7 : Pointeurs en langage C

2. Déclaration et initialisation d'un pointeur 2.1 Déclaration

Remarques :

- A la déclaration d'un pointeur p, il ne pointe a priori sur aucune variable précise : p est un pointeur non initialisé.

  ⚠ Toute utilisation de p devrait être précédée par une initialisation.

- la valeur d'un pointeur est toujours un entier (codé sur **16 bits**, **32 bits** ou **64 bits**). le type d'un pointeur dépend du type de la variable vers laquelle il pointe. Cette distinction est indispensable à l'interprétation de la valeur d'un pointeur. En effet

  Pour un pointeur sur une variable de type char, la valeur donne **l'adresse de l'octet** ou cet variable est stockée.

  Pour un pointeur sur une variable de type short, la valeur donne **l'adresse du premier des 2 octets** où la variable est stockée

  Pour un pointeur sur une variable de type float, la valeur donne **l'adresse du premier des 4 octets** où la variable est stockée

CHAPITRE 7 : Pointeurs en langage C

2. Déclaration et initialisation d'un pointeur

2.2 Initialisation

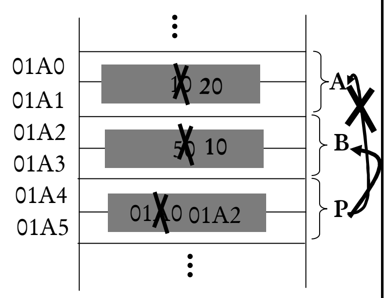
Pour initialiser un pointeur, le langage C fournit l'opérateur unaire **&**. Ainsi pour récupérer l'adresse d'une variable A et la mettre dans le pointeur P (P pointe vers A) :

P = & A

Exemple 1 :

int A, B, *P; /*supposons que ces variables occupent la mémoire à partir de l'adresse 01A0 */

A = 10;
B = 50;
P = &A ; // se lit mettre dans **P** l'adresse de **A**
B = *P ; /* mettre dans **B** le contenu de de la variable pointe par **P***/
P = 20;** /*mettre la valeur **20** dans la variable pointé par **P/
P = &B; // **P** pointe sur **B**



CHAPITRE 7 : Pointeurs en langage C

2. Déclaration et initialisation d'un pointeur

Exemple 2 :

```
#include <stdio.h>
#include <conio.h>

Void main( )
{ float  a , *p;  /*supposons que ces variables sont représentées
                   en mémoire à partir de l'adresse 01BF*/
   clrscr( ); // pour effacer l'écran → <conio.h>
   p = &a;
   printf("Entrer une valeur :");
   scanf("%f",p);  // on saisie la valeur 1.4

   printf("\nAdresse de a = %x  Contenu de a = %f",p,*p);
   *p += 0.4;
   printf("a = %f        *p = %f " , a,*p);
   getch( ); // pour lire un caractère → <conio.h>
}
```

01BF
01C0
01C1
01C2
01C3
01C4
~~1.4~~ 1.8
01BF
A
P

CHAPITRE 7 : Pointeurs en langage C

3. Opérations élémentaires sur les pointeurs

- L'opérateur & : 'adresse de' : permet d'obtenir l'adresse d'une variable.
- L'opérateur * : 'contenu de' : permet d'accéder au contenu d'une adresse.
- Si un pointeur P pointe sur une variable X, alors *P peut être utilisé partout où on peut écrire X.
- <u>**Exemple :**</u> int X=1, Y, *P Après l'instruction, **P = &X** ;

  On a :

Y = X + 1	équivalente à	Y = *P + 1
X += 2	équivalente à	*P += 2
++X	équivalente à	++ *P
X++	équivalente à	(*P)++

CHAPITRE 7 : Pointeurs en langage C

3. Opérations élémentaires sur les pointeurs (suite)

- Le seul entier qui puisse être affecté à un pointeur d'un type quelconque P est la constante entière 0 désignée par le symbole NULL défini dans <stddef.h>.

  On dit alors que le **pointeur P ne pointe 'nulle part'.**
- Exemple :

```
#include <stddef.h>
...
int  *p,  x , *q;
short  y = 10 , *pt = &y;
p = NULL ;  /* Correct */
p = 0 ;          /* Correct */
x = 0 ;
p = x ;        /* Incorrect ! bien que x vaille 0 */
q = &x ;
p = q ;        /* Correct : p et q pointe sur des variables de même type*/
p = pt ;       /* Incorrect : p et pt pointe sur des variable de type différent */
```

CHAPITRE 7 : Pointeurs en langage C

Exercices

Trouvez les erreurs dans les suites d'instruction suivantes :

a) **int *p , x = 34; *p = x;**
   ***p = x est incorrect parce que le pointeur p n'est pas initialisé**

b) **int x = 17 , *p = x; *p = 17;**
   ***p = x est incorrect. Pour que p pointe sur x → *p = &x**

c) **double *q; int x = 17 , *p = &x; q = p;**
   q = p incorrect. q et p deux pointeurs sur des types différent

d) **int x, *p; &x = p;**
   &x = p incorrect. &x n'est pas une variable (lvalue) et par conséquent ne peut pas figurer à gauche d'une affectation.

e) **char mot[10], car = 'A', *pc = &car ; mot = pc;**
   mot = pc incorrect. mot est un pointeur constant et on ne peut pas changer sa valeur. Ce n'est pas une variable (lvalue).

CHAPITRE 7 : Pointeurs en langage C

4. Pointeurs et Tableaux

- En C, il existe une relation très étroite entre tableaux et pointeurs. Ainsi, chaque opération avec des indices de tableaux peut aussi être exprimée à l'aide de pointeurs.

- Comme déjà mentionné (au chapitre 6), le nom d'un tableau représente l'adresse de son premier élément :
  - Tableau à un dimension (int T[N]) :
    - le nom T du tableau est un pointeur constant sur le premier élément (1er entier) du tableau
    - T et &T[0] contiennent l'adresse du premier élément (1er entier) du tableau.

  - Tableau à deux dimensions(int T[N][M]) :
    - le nom T est un pointeur constant sur le premier tableau (d'entiers).
    - T[i] est un pointeur constant sur le premier élément (1er entier) du ième tableau.
    - T et T[0] contiennent la même adresse mais leur manipulation n'est pas la même puisqu'ils ne représentent pas le même type de pointeur.

4.1 Adressage et accès aux composantes d'un tableau à une dimension

- En déclarant un tableau A de type int (int A[N]) et un pointeur P sur des variables entière (int *P),
- l'instruction P = A crée une liaison entre le pointeur P et le tableau A en mettent dans P l'adresse du premier élément de A (de même P = &A[0]).
- A partir du moment où P = A, la manipulation du tableau A peut se faire par le biais du pointeur P. En effet :

p	pointe sur	A[0]	*p	désigne A[0]
p+1	pointe sur	A[1]	*(p+1)	désigne A[1]
….				
p+i	pointe sur	A[i]	*(p+i)	désigne A[i]

où $i \in [0 , N-1]$

CHAPITRE 7 : Pointeurs en langage C

4. Pointeurs et Tableaux

4.1.1 Exemple (Lecture et Affichage d'un tableau matérialisé par un pointeur)

```
#include <stdio.h>
#define N 10

void main()
{
  float t[N] , *pt ;
  int i ;

  printf("Entrez %d entiers\n", N) ;

  pt = &t[0] ;      /* ou pt  = t */
  for (i = 0 ; i<N; i++)
    scanf("%f", pt+i) ;  /* pt+i pointe sur A[i] */

  printf("\nTableau lu : \n") ;
  for (i = 0 ; i<N ; i++)
    printf("%7.2f", *(pt+i)) ; /* *(pt+i)
équivalente à pt[i] */
}
```

```
/* Autre Solution sans déclarer la variable i */
#include <stdio.h>
#define N 10

void main()
{
  float T[N] , *pt ;

  printf("Entrez %d entiers\n", N) ;

  for (pt = T ; pt<T+N; pt++)
    scanf("%f", pt) ;

  printf("\nTableau lu : \n") ;

  for (pt = T ; pt<T+N; pt++)
    printf("%7.2f", *pt) ;
}
```

CHAPITRE 7 : Pointeurs en langage C

4. Pointeurs et Tableaux

4.2 Adressage et accès aux composantes d'une matrice

- En déclarant une matrice A de type int (int A[M][N]) et un pointeur P sur des variables entières (int *P),
- l'instruction P = A[0] crée une liaison entre le pointeur P et la matrice A en mettent dans P l'adresse du premier élément de la première ligne de la matrice A (P = &A[0][0]).
- A partir du moment où P = A[0], la manipulation de la matrice A peut se faire par le biais du pointeur P. En effet :

p	pointe sur	A[0][0]	et	* p	désigne A[0][0]
p + 1	pointe sur	A[0][1]	et	* (p + 1)	désigne A[0][1]
….					
p + N	pointe sur	A[1][0]	et	* (p + N)	désigne A[1][0]
p + N + 1	pointe sur	A[1][1]	et	* (p + N + 1)	désigne A[1][1]
….					
p + i * N + j	pointe sur	A[i][j]	et	* (p + i * N + j)	désigne A[i][j]

où $i \in [0 , M-1]$ et $j \in [0 , N-1]$.

CHAPITRE 7 : Pointeurs en langage C

4. Pointeurs et Tableaux

4.2.1 Exemple (Lecture et Affichage d'une matrice matérialisé par un pointeur)

```
#include <stdio.h>
#define M 4
#define N 10

void main()
{
 short A[M][N] ;
 short *pt ;
 int i, j ;

 /* lecture d'une matrice */
 pt = &A[0][0] ;/* ou bien pt = A[0] ; */
 for (i = 0 ; i<M ; i++)
 {
    printf("\t ligne n° %d\n", i+1) ;
    for (j = 0 ; j<N ; j++)
      scanf("%i", pt + i * N + j ) ;
 }
 for (i = 0 ; i<M ; i++)
 {
    for (j = 0 ; j<N ; j++)
      printf("%d", *( pt + i * N + j ) ) ;
    printf("\n") ;
 }

}
```

CHAPITRE 7 : Pointeurs en langage C

4. Pointeurs et Tableaux

4.3 Arithmétiques des pointeurs

Affectation par un pointeur sur le même type :

- Soient P1 et P2 deux pointeurs sur le même type de données.
- L'affectation : P1 = P2 ; fait pointer P1 sur le même objet que P2.

Addition et soustraction d'un nombre entier :

- Si P pointe sur l'élément A[i] d'un tableau, alors :
- P+n *pointe sur* A[i+n] et P-n *pointe sur* A[i-n]

Incrémentation et décrémentation d'un pointeur :

- Si P pointe sur l'élément A[i] d'un tableau, alors après l'instruction :
- P++ ; P *pointe sur* A[i+1]
- P += n ; P *pointe sur* A[i+n]
- P-- ; P *pointe sur* A[i-1]
- P -= n ; P *pointe sur* A[i-n]

Comparaison de deux pointeurs :

- On peut comparer deux pointeurs ***de même type*** par : <, >, <=, >=, == ou !=
- La comparaison de deux pointeurs qui pointent dans le même tableau est équivalente à la comparaison des indices correspondants.



4.4 Autres déclarations des pointeurs :

En C, il existe d'autres déclarations des pointeurs. En effet :

- **Tableau de pointeurs :**
  int *Tab[20] ;
  déclare un tableau Tab de 20 pointeurs d'entiers.

- **Pointeur de tableaux :**
  int (*pt)[30] ;
  déclare un pointeur pt sur des tableaux de 30 composantes.

- **Pointeur de pointeurs :**
  int **pt ;
  déclare un pointeur pt qui pointe sur des pointeurs d'entiers.

5. Allocation dynamique

- La déclaration d'un tableau définit un tableau "statique" (il possède un nombre figé d'emplacements). Il y a donc un gaspillage d'espace mémoire en réservant toujours l'espace maximal prévisible.

- Il serait souhaitable que l'allocation de la mémoire dépend du nombre d'éléments à saisir. Ce nombre ne sera connu qu'à l'exécution : c'est l'allocation dynamique.

CHAPITRE 7 : Pointeurs en langage C

5.1 Fonctions d'allocation dynamique de la mémoire

- En C, il existe 4 fonctions pour gérer l' allocation dynamiquement de la mémoire

Bibliothèque <**stdlib.h**>	
`char *malloc(taille)`	allocation d'un bloc
`char *calloc(taille, sizeof(type))`	allocation & initialisation d'un bloc
`char *realloc(char *, taille)`	modification de la taille d'un bloc
`void free(char *)`	libération d'un bloc

- Chacune des fonctions malloc, calloc ou realloc, prend une zone d'une taille donnée dans l'espace mémoire libre réservé pour le programme (appelé ***tas* ou *heap***) et affecte l'adresse du début de la zone à une variable pointeur.

- S'il n'y a pas assez de mémoire libre à allouer, la fonction renvoie le pointeur NULL.

CHAPITRE 7 : Pointeurs en langage C

5.2 Fonctions malloc et free

- **5.2.1 malloc**

<pointeur> = <type> malloc(<taille>);

- **<type>** est un type pointeur définissant la variable pointé.par <pointeur>
- **<taille>** est la taille, en octets, de la zone mémoire à allouer dynamiquement. <taille> est du type **unsigned int**, donc on ne peut pas réserver plus de 65536 octets à la fois
- La fonction **malloc** retourne l'adresse du premier octet de la zone mémoire allouée. En cas d'échec, elle retourne **NULL**.

- **5.2.2 free**

- Si on n'a plus besoin d'un bloc de mémoire réservé dynamiquement par **malloc**, alors on peut le libérer à l'aide de la fonction **free**.

free(<pointeur>);

- Libère le bloc de mémoire désigné par le pointeur <pointeur>

CHAPITRE 7 : Pointeurs en langage C

<u>5.2.3 Exemple</u> (Allocation dynamique, Saisie et Affichage d'un tableau)

```
#include <stdio.h>
#include <stdlib.h>

int  main()
{
   short  *pt;
   int N , i;
   printf("Entrez la taille N  du tableau \n") ;
   scanf("%d", &N) ;

   pt = ( short * ) malloc( N * sizeof( short ) );
   if (pt == NULL)
   {
     printf("Mémoire non disponible") ;
     system("pause") ;
     return 1;
   }
```

```
   printf("Saisie du tableau : ");
   for (  i = 0  ; i < N; i++)
        scanf("%d", pt + i ) ;

   printf("Affichage du tableau ") ;
   for ( i= 0  ; i < N; i++)
         printf("%d\t", *( pt + i ) ) ;

   free( pt );

   return 0;
}
```